687 Final Project Report

Vinay Ramesh, Ignacio Gavier

December 2021

Introduction

E-commerce is a field that's been gaining increasing importance and prominence in recent years, especially with events like the 2020 pandemic forcing everyone inside. As such, we've started to see significantly more industries shift towards an online retail methodology. One of these industries, the fashion industry, specifically has some unique challenges when it comes to the online space.

The fashion industry has, as of recently, almost wholly shifted to a 'Fast Fashion' mindset. In this state, clothes are produced fast, sold fast, and delivered fast, creating cycles of clothes that are in-season and out-of-season. However, this often leads to large warehouses full of clothes inventory just sitting around unused, in these cases, they're often sold off at significantly cheaper prices in clearance sales. In these cases, they're discounted heavily to try and incentivize as many consumers as they can to clear out the inventory so that the company can generate as much of their money that they spent on the clothes back as possible. It's this problem of optimizing the discount applied to clearance sales that we will be tackling in this project report. Specifically, we model the problem of clearance sales as a Markov Decision Process, and apply three different reinforcement learning methods - SARSA, Q Learning, and Actor Critic - to it, and from this we will introduce the results we obtained and analyze how these agents performed on the problem.

Naturally, some abstractions have to be made in order to turn what is otherwise an incredibly complex problem with hundreds of factors into something that can be represented as a simple MDP. These will be discussed in further detail, since some understanding of the complex problem of optimizing clearance sales returns is required to understand the generalizations made for the MDP.

A Broader Look at Clearance Sales

There is a lot that goes into quantifying exactly how items are processed and discounts are applied during clearance sales, and each and every single item has completely different behavior. To understand the generalizations we make, we need to look at the three main types of articles that are sold in stores, and how their **demand** works. Specifically, these three types of items are:

- Never Out of Stock (NOOS) items: these are items that are timeless, and will never truly be out of season, things like plain white tshirts, or a nice pair of jeans. The demand for these items is heavily elastic. What that means is that for a small change in price, you have an inversely proportional, large, change in demand. This is because there is always demand for them, and as such any small change in price pushes people who were going to buy the products anyways to buy early. Branded products also fall under this umbrella, something like a fancy pair of Nike shoes is never going out of style or fashion so its demand behaves the same as other NOOS items.
- Non-Seasonal Items (NSI): These are items that were in season at some point in the recent past, but are no longer in season for some reason or another. This can include things like winter socks in the spring, or just weird things (like maybe glow in the dark shirts were the fashion 2 seasons ago). Demand for these tends to be uniform regardless of discount, but follows a "step" pattern, where the demand might be uniform from say, 10% till 20% discount, but then increase and stay uniform from 20% till 40% discount.
- Miscellaneous items (misc): These items are complete wild cards. Nobody can really predict how their demand is going to work, and discounts have a completely random effect on their demand. These generally tend to be weird trinkets and other paraphernalia like pins and ties.

The curves below show how the demand of each of these items might grow with demand:

Figure 1: Random demand vs Discount for misc items

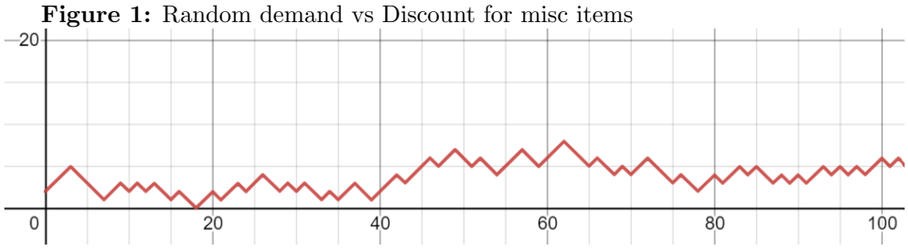


Figure 2: Step-like demand vs. Dicsount for NIS items

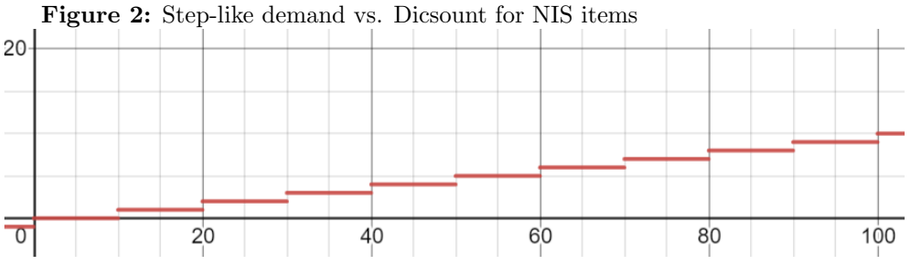
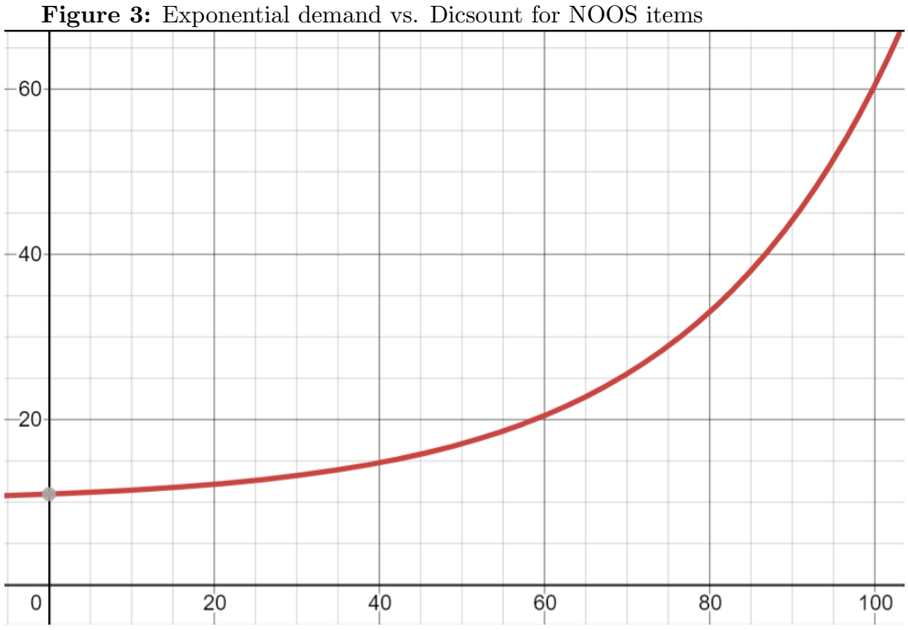


Figure 3: Exponential demand vs. Dicsount for NOOS items



What we notice is that the 3 have very different demand curves, however we can somewhat normalize these if we notice a few things. The first is that we can completely disregard misc items, they are fully unpredictable, and as such we only need to look at NOOS and NSI items. In this case, we notice that they both do actually follow a pattern in which their demand does increase overall with the discount offered. As such, when we're estimating how a price change affects the demand, we can normalize these curves on the principle that as price increases, our demand decreases, and vice versa. This brings us to the next major problem of act

Quantifying Demand

Coming up with a generalized demand curve is one of the hardest parts of creating a model for clearance sales. There's a few constraints that need to be followed, those being:

- Demand is inversely proportional to price. As the price decreases, the demand will increase
- Demand is linearly proportional to the time since the start of the sale. The longer a sale has gone on, assuming that size availability is constant, the demand for a product will increase.
- Customers have weird psychology about availability of supply. This is very hard to quantify exactly, but basically the presumed supply of a product will affect how much demand is perceived of it. It's easy to visualize this with scarce items, there is a higher demand for limited edition items. As such we introduce a parameter, ω that represents this, which will be gone on in further detail later.
- An elasticity parameter, α is generated based on the perceived elasticity of the product. This is a simple constant multiplied with the demand to represent just how elastic it is, i.e. how much of a change in demand there is when one of the variables is changed. A higher elasticity parameter means a higher change in demand at every step.

Keeping these in mind, we can start to generate curves that fit these 3 constraints. One important thing to note is the exact importance that each aspect of demand is being given. For example, if we give more importance to time, then our reductions in price don't matter as much. Likewise, if we place more importance on price, then the time till end of sale doesn't matter as much anymore. Combining both of these together, what we end up getting is twofold. First we have an Inventory Function $I(t)$, this takes in the current time step, and outputs how much inventory is controlled at that state, i.e. how much of the inventory the company is willing to actually sell off at any given time. As such, this gives us an inventory function of:

$$I(t) = \alpha \frac{t}{L} * I_{max}$$

where L is the maximum time step, and I_{max} is the starting inventory. Next up, looking at the price-time relation, a few different functions were tested. At first, a logarithmic function conditioned on t and the log of the price was used, but this ended up being far too reliant on t, and not as reliant on price. Secondly, an exponential decay function was used, but this gave very mixed results, decreasing inventory a bit too slowly for it to be of any use. As such, we finally settled on a linear function for the demand. This lies in between the step function of NSIs and the heavily exponential function of NOOS items, and as such was a natural fit. The function takes a form similar to:

$$\omega f(t) * g(p)$$

Where p is the price. As such, it just came down to identifying what f, g were. Through emperical testing, t was still overpowering in the function, and as such was toned down by applying a logarithmic function to it, giving us

$$\omega log(t) * g(p)$$

For g, it really came down to a lot of tinkering. We initially started with a naive estimate of g(p), given by:

$$\frac{1}{p}$$

But in this case, we found that the price had no actual effect on the overall reward, because when we multiplied it by price to get the reward, the factors cancelled out, ensuring that the final value was conditioned only on time. To try and remedy this, we tried using an exponential function for $g(p)$, giving us:

$$\frac{1}{p_{max} * e^{-2 * \frac{p}{p_{max}}}}$$

In this case, we ended up with a very weird amalgam for our demand curve, which was eventually scrapped because the same problem ended up happening, where it ended up cancelling with the price that was multiplied for our reward, and as such time became the defining factor for demand rather than price. To finally fix this, we decided to make the dependence on price linear. To do so, instead of keeping price in the denominator of the function, it was transferred to the numerator and a function was placed there to still give a greater value if the price was lower. What we ended up getting was a slightly offbeat linear function represented as:

$$g(p) = \frac{\frac{-5p}{p_{max}} + 6}{p_{max}}$$

where p_{max} is the maximum possible price. The constant numbers in here were done to ensure that the factor of price could never be 0. Putting it all together, we get:

$$D(p,t) = \alpha \frac{t}{L} I_{max} \times \frac{\omega log(t) * \frac{-5p}{p_{max}} + 6}{p_{max}}$$

This is the function that determines our state transitions. With this in mind, we can go into how the state representation for the MDP was set up.

Creating a State Representation

Intuitively, the actions that the agent should be taken are conditioned on 3 things: The current inventory, the current discount being offered, and the current time. As such, we can very easily translate this into our state space. However, in real world situations, this might end up proving to create *massive* state spaces that are unreliable to work with, as such we have to further discretize an already discrete state space. Thankfully, this problem is already a solved one. Fashion companies classify the price ranges and inventory ranges into a few categories depending on a few factors, which are covered below:

- For Inventory, the ranges are split 4-ways, based on the risk of selling out prematurely. These ranges are No risk, Low Risk, Medium Risk, and High Risk. Each risk level represents a percentage range of the maximal inventory, with No risk being 100% - 80%, Low risk being 80% - 60%, medium risk being 60% - 40%, and high risk being 40% - 0% of the maximum inventory. These are the 4 states that our inventory is conditioned on.
- For Price, we apply discounts in flat ranges of 5. The reasoning for this is because of consumer psychology, since it is easier for a consumer to process discounts that are multiples of 5, even if it's not necessarily an optimal answer. As such, the discount ranges are split into ranges of 5%, going from a 0% discount to a 80% discount. The reason we don't go up to 100% is because items given out for free are not making money, so the industry tends to put a hard cap on discount applied for clearance items, which is generally around 80-90%.

Given the above, we have one last thing to consider: time. The sale will always end at a predetermined time period, and as such the MDP is intrinsically conditioned on the time. To account for this, the time was introduced into the state space, giving us a final state representation of (I, p, t). Additionally, we made the MDP finite horizon, any state where $t = L$ is a terminal state by default, and at every step the value of t is incremented by one (i.e. whenever the agent takes an action, it will move them to a state in the next time step). The time steps are not discretized further, since our decisions do have to be made at every individual time step t. Finally, our initial state for validation is always the same, we start with maximal inventory, at a discount of 0% (or 100% of the base price), and at a time of 0, giving us a fixed d_0 of $(I_{max}, 100\%, 0)$. Given all of this, we can finally get to our actual MDP formulation.

MDP Properties

For any MDP, there's a few properties that we need to know, those being: d_0 the initial state, R the reward function, S the state space, p the transition function, A the set of actions available, and γ the discount parameter. Below I will define all the specific parameters for the fast fashion MDP and the reasoning behind each of the choices made.

- S, the state space, was defined earlier. We use a 3D vector $(I_{range}, p_{range}, t)$ to represent each state, where I_{range} is the risk level of the current inventory, p_{range} is the range of the current discount, and t is the current time step.
- d_0 the initial state, is always $(I_{max}, 100\%, 0)$, representing us starting with maximum inventory, no discount, and a time of 0 (random initialization was used in training the algorithms, however).
- A is a set of discounts that can be added to the price, we set the discounts to a flat $[-10, 0, 10, 20]$. What this means is that the agent can either remove 10% from the discount, keep it the same, or add 10% or 20% to the discount. In the actual fashion space, the discount cannot be removed, i.e. if we have a 50% discount at time 3, then at time 4 we cannot bring it down to a 45% discount. However for the purposes of the MDP, more unique actions were allowed in order to let the agent make more decisions and not pigeonhole itself into selling out its inventory too quickly.
- p is defined by the demand function. The actual probabilities are a bit tricky to calculate, but for the most part the inventory and price declines are stochastic. This is because the inventory is set up to be ranges, so what output state we get is determined on the current inventory, however it's not random, because our demand is only based on price and time, so as such if we take action a in state s, then we will move to state s' with a fixed probability that is based on the maximum inventory put in. As such, p changes based on the initialization of the MDP, but it is a consistent stochastic transition function for the inventory. Transitions between price states and time states are fully deterministic.
- R, the reward function, is very simple. For any given state s and action a, the reward is going to be the amount of items sold $\times$ the price they were sold at.
- γ, the discount parameter, was set to a solid 0.95 throughout. This was calculated somewhat empirically, since we don't really want the agent to be prioritizing immediate rewards that much, since it's valuable to try and sell items at all stages of time.

All of these factors together form the MDP that we trained the RL algorithms on. In the next section, we go into further detail on the algorithms used and how they performed on this MDP representation of the problem set.

Algorithms

The RL algorithms we used to solve the problem were SARSA –Table 1–, Q-Learning –Table 2– and One-Step Actor-Critic –Table 3–. The first two algorithms are similar and both are based on storing a tabular action-value function. The main difference between them is that SARSA updates the action-value function towards the values that are best for the current policy (see step 10 in 1), while Q-Learning directly updates the values towards the policy that is locally optimal (see step 9 in 2). Therefore, SARSA is called *on-policy* and Q-Learning *off-policy*. Both algorithms have similar structure in the sense that they use tabular values, but Q-Learning is usually faster for convergence due to the difference already mentioned. In the MDP we are trying to solve, we set the states to be discrete so they can be processed by these two algorithms.

On the other hand, the One-Step Actor-Critic algorithm is a completely different approach since it uses two parameterized models, actor and critic, for the estimation of the policy and state-value functions. The first model outputs an action given a state, and the second model outputs a state-value given a state. Both models have trainable parameters and they are updated at each step of the episode considering only the observations in the given step. In contrast with the other two algorithms, One-Step Actor-Critic can be used for continual states or actions because it relies on the models parameters to compute actions and values. In the MDP we are trying to solve, this is not a problem since the states, although they are discrete, are intrinsically continual, i.e., they are not categorical. Consequently, if

after training the discretization was removed then the agent would still work with similar performance, while the agent trained with SARSA or Q-Learning would need an interpolation of the tabular policy.

Exploration. For each step of SARSA and Q-Learning, actions are sampled directly from the action-value functions. This requires a method for converting tabular values into probabilities. In this work we used the *softmax* method to sample actions:

$$\pi(a_i|s) = \frac{e^{Q(s,a_i)/\sigma}}{\sum_{a_j} e^{Q(s,a_j)/\sigma}},$$

where σ is a hyperparameter called *exploration*. As $\sigma \to 0$, $\pi(a|s) \to \arg\max_{a'} Q(s,a)$, which means that the agent follows a deterministic policy. While as $\sigma \to \infty$, $\pi(a|s) \to 1/|\mathcal{A}|$, which means that the agent follows a uniform random policy. The value of this hyperparameter was tuned and is described in the next section. We allowed the option to schedule a decay, following:

$$\sigma \leftarrow \sigma/2,$$

every E_σ episodes, which is a (slow-)exponential decay. The value of E_σ was also tuned and is described in the next section.

Updates. The three algorithms use hyperparameters to update the values to compute the final policy. In the case of SARSA and Q-Learning, the updates are done over the action-values, while in the case of One-Step Actor-Critic, the updates are done over the state-values and the policy. The value of these hyperparameters were tuned and are described in the next section. We also allowed the option to schedule a decay for each one of them, following:

$$\alpha_{\mathbf{w},\boldsymbol{\theta}} \leftarrow \alpha_{\mathbf{w},\boldsymbol{\theta}}/2,$$

every $E_{\mathbf{w},\boldsymbol{\theta}}$ episodes, which is a (slow-)exponential decay. The values of $E_{\mathbf{w},\boldsymbol{\theta}}$ were also tuned and are described in the next section.

Tabular Q. The action-value tables for SARSA and Q-Learning were initialized with random values following a normal distribution. This initialization should not affect the final performance of the algorithms.

Actor and Critic. For the models in One-Step Actor-Critic, we used fully-connected neural networks with ReLU activations. The input for both of them is a three-dimensional state that is zero-shifted and normalized to one, and then processed by the network. The output of the actor has as many neurons as number of possible actions are in the MDP, and they are processed with a numerically-stable LogSoftMax layer, representing the log-probabilities of each action:

$$\ln \pi(a_i|s,\boldsymbol{\theta}) = z_A(a_i|s,\boldsymbol{\theta}) - \bar{z}_A(s,\boldsymbol{\theta}) - \ln \sum_{a_j} e^{z_A(a_j|s,\boldsymbol{\theta}) - \bar{z}_A(s,\boldsymbol{\theta})},$$

where $z_A(a_i|s,\boldsymbol{\theta})$ are the logits of the last fully-connected layer (without ReLU) of the actor and $\bar{z}_A(s,\boldsymbol{\theta}) = \max_{a_i} z_A(a_i|s,\boldsymbol{\theta})$. The output of the actor, however, is a single neuron:

$$\hat{v}(s|\mathbf{w}) = z_C(s,\mathbf{w}),$$

where $z_C(s,\boldsymbol{\theta})$ is the logit of the last fully-connected layer (without ReLU) of the critic. The weights in all of the layers were initialized randomly following the default uniform distribution implemented in PyTorch.

State initialization. During training, for the three algorithms we used a uniformly random initialization of the state (including the temporal dimension) to boost the learning. During validation/testing, we initialized it as deterministic in the maximum price and maximum inventory.

Hyperparameter tuning

For the MDP, we selected $\gamma = 0.95$, $L = 12$ and the number of episodes for each algorithm was 2000. SARSA and Q-Learning algorithms have the same hyperparameters, while One-Step Actor-Critic has some of the hyperparameters used by the other two, but also has extra hyperparameters. In order to find optimal hyperparameters for each algorithm, we followed the following steps:

- Coarse tuning. We selected a wide range of hyperparameters and selected the best results in terms of discounted return.
- Fine tuning. We explored more narrow ranges of hyperparameters in the vicinity of the hyperparameters obtained in the coarse tuning. Then we selected the best of each algorithm. The results are presented based on this selection.

This is summarized in Table 1. The agents with the selected hyperparameters were trained over 5000 episodes.

Algorithm 1: SARSA for estimating $\pi \approx \pi_*$.

Input: a tabular action-value function $Q(s, a)$
Parameters: step size $\alpha_{\mathbf{w}} > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$ `// If s is terminal, then` $Q(s, \cdot) = 0$
for episode $\in 1, 2, \ldots$ **do**
  Initialize S `// First state of episode`
  Choose A from S using policy derived from $Q(S, \cdot)$ `// Softmax`
  for $t \in 0, 1, \ldots, L-1$ **do**
    Take action A, observe S', R
    Choose A' from S' using policy derived from $Q(S, \cdot)$ `// Softmax`
    $\delta = R + \gamma Q(S', A') - Q(S, A)$
    $Q(S, A) \leftarrow Q(S, A) + \alpha_{\mathbf{w}} \delta$
    $S \leftarrow S'$
    $A \leftarrow A'$
$\pi(a|s) = \arg\max_{a'} Q(s, a')$
Output: π

Algorithm 2: Q-Learning for estimating $\pi \approx \pi_*$.

Input: a tabular action-value function $Q(s, a)$
Parameters: step size $\alpha_{\mathbf{w}} > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$ `// If s is terminal, then` $Q(s, \cdot) = 0$
for episode $\in 1, 2, \ldots$ **do**
  Initialize S `// First state of episode`
  for $t \in 0, 1, \ldots, L-1$ **do**
    Choose A from S using policy derived from $Q(S, \cdot)$ `// Softmax`
    Take action A, observe S', R
    $\delta = R + \gamma \arg\max_a Q(S', a) - Q(S, A)$
    $Q(S, A) \leftarrow Q(S, A) + \alpha_{\mathbf{w}} \delta$
    $S \leftarrow S'$
$\pi(a|s) = \arg\max_{a'} Q(s, a')$
Output: π

Algorithm 3: One-Step Actor-Critic for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$.

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s|\mathbf{w})$
Parameters: step sizes $\alpha_{\boldsymbol{\theta}} > 0, \alpha_{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d'}$
for episode $\in 1, 2, \ldots$ **do**
  Initialize S `// First state of episode`
  $I = 1$
  for $t \in 0, 1, \ldots, L-1$ **do**
    Choose A from S using $\pi(\cdot|S, \boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta = R + \gamma \hat{v}(S'|\mathbf{w}) - \hat{v}(S|\mathbf{w})$ `// if` S' `is terminal, then` $\hat{v}(S'|\mathbf{w}) = 0$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha_{\mathbf{w}} \delta \nabla \hat{v}(S|\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S, \boldsymbol{\theta})$
    $I \leftarrow I\gamma$
    $S \leftarrow S'$
Output: $\pi_{\boldsymbol{\theta}}$

Results

Metrics

Before we talk about the results that the algorithms achieved, what's important to note is the metrics that we are judging them by. There's 3 main metrics that we use to actually judge how good our agents are performing.

Random Agent

Juxtaposed on the median return plots alongside our agent is an agent that takes wholly random actions at every time step, the normal agent is portrayed in black, and the random agent is portrayed in pink. The reasoning a random agent was chosen is to show a control group as to what the reward looks like if the agent makes random actions, and is used to put into light how our trained agent

	Hyperparameter	Symbol	Coarse Range	Fine Range	Best
SARSA	Exploration	σ	$[10^{-3}; 10^{3}]$	$[10^{-1.5}; 10^{-1}]$	10^{-1}
	Exploration decay	E_{σ}	$\{200; 500; L\}$	$\{200; 500; L\}$	L
	Update table	$\alpha_{\mathbf{w}}$	$[10^{-5}; 10^{-1}]$	$[10^{-2.5}; 10^{-2}]$	10^{-2}
	Update table decay	$E_{\mathbf{w}}$	$\{200; 500; L\}$	$\{200; 500; L\}$	L
QL	Exploration	σ	$[10^{-3}; 10^{3}]$	$[10^{-1.5}; 10^{-1}]$	10^{-1}
	Exploration decay	E_{σ}	$\{200; 500; L\}$	$\{200; 500; L\}$	L
	Update table	$\alpha_{\mathbf{w}}$	$[10^{-5}; 10^{-1}]$	$[10^{-2.5}; 10^{-2}]$	10^{-2}
	Update table decay	$E_{\mathbf{w}}$	$\{200; 500; L\}$	$\{200; 500; L\}$	L
OSAC	Update actor	$\alpha_{\boldsymbol{\theta}}$	$[10^{-5}; 10^{-1}]$	$[10^{-2.5}; 10^{-1.5}]$	10^{-2}
	Update actor decay	$E_{\boldsymbol{\theta}}$	$\{200; 500; L\}$	$\{200; 500; L\}$	L
	Hidden layers actor	$H_{\boldsymbol{\theta}}$	$\{0; 1; 2\} \times \{[20; 50; 100]\}$	$\{0; 1\} \times \{[30; 40; 50]\}$	[]
	Update critic	$\alpha_{\mathbf{w}}$	$[10^{-5}; 10^{-1}]$	$[10^{-2.5}; 10^{-1.5}]$	10^{-2}
	Update critic decay	$E_{\mathbf{w}}$	$\{200; 500; L\}$	$\{200; 500; L\}$	L
	Hidden layers critic	$H_{\mathbf{w}}$	$\{0; 1; 2\} \times \{[20; 50; 100]\}$	$\{1; 2\} \times \{[30; 40; 50]\}$	[40]

Table 1: Hyperparameter exploration.

behaves and why it behaves the way it does.

Maximum Theoretical Return

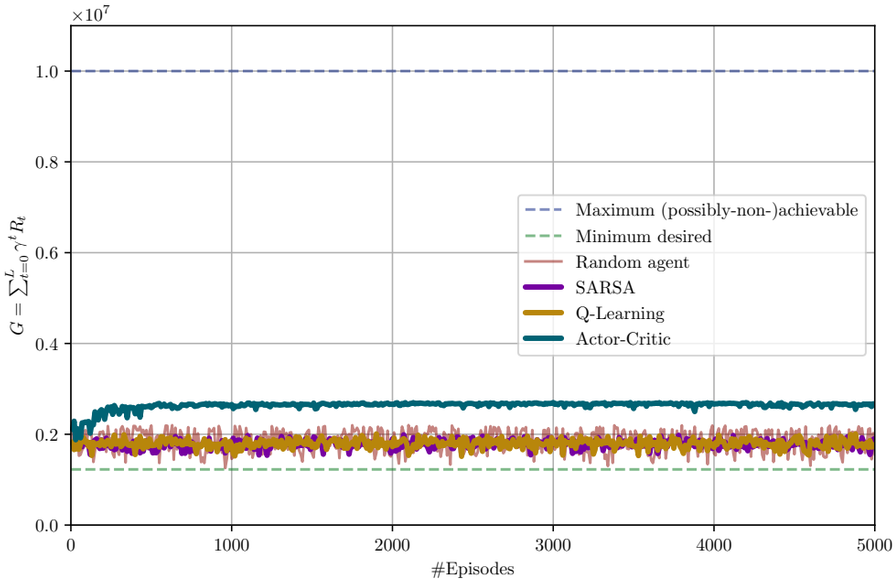
The 2^{nd} metric we use is the maximal possible return from any given episode. This is a very simple number to calculate, since the theoretical maximum return we can get is $I_{max} \times p_{max}$, and as such this is placed on the graph as well. One thing to note is that this maximal limit is practically never achievable, and in fact it's not even close to achievable practically. The reasoning is because in real world situations (and with our agent) we want to decrease the price so that our demand increases, and as such we actually sell out of inventory. As such this is just being used to show off the upper bound of our agent's theoretical performance.

Ideal Expected Return

Finally, the last metric we have is probably the most important one. As a general rule of thumb when creating an environment for sales predictions, companies have an expectation for how much product they will sell at any time step t. In addition, it is assumed that the price also decreases at a standard rate. What this results in is a slight bell curve where the ideal minimum amount we're making peaks somewhere midway through the sale period, and tapers off at the start and towards the end. The reason we use this as a metric is because it represents how much we want to ideally be selling without putting much thought into things, it's a "uniform" decrease in price and inventory. As such, we keep this (it appears as a line on the chart since the bell curve is per-episode, which ends up looking like a straight line when scaled out to thousands of episodes on a graph) metric as something our agent should always try to be at least equal to, and ideally ahead of on all counts. Surpassing this metric is the most important one for the agent.

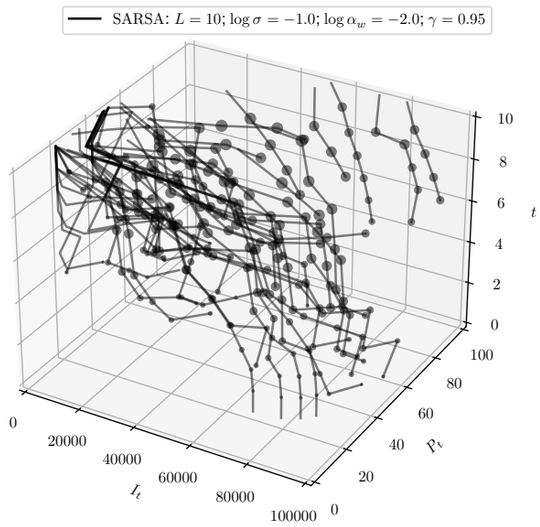
Median Results Graph

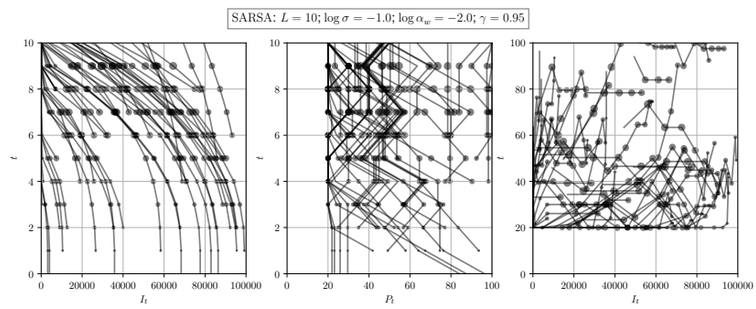
Below we can see the median results graph for all 3 agents when run 10 experiments on the MDP. The exact curve and their meanings will be discussed in the specific sections for each agent.



SARSA

One thing we can notice right off the bat is that SARSA (and also QLearning) do not have the strongest performance. In particular, we can see for sarsa that it performs very erratically, almost exactly the same as the random agent. However it does manage to consistently stay above our minimum desired, which is still a good thing regardless. The reason for this is a bit complex, but likely has to do with the fact that the state space is naturally very large and does not allow for backtracking. As such, sarsa can only ever learn through the trajectories it takes and cannot come back to any individual state. The trajectory plotting specifically are shown below:

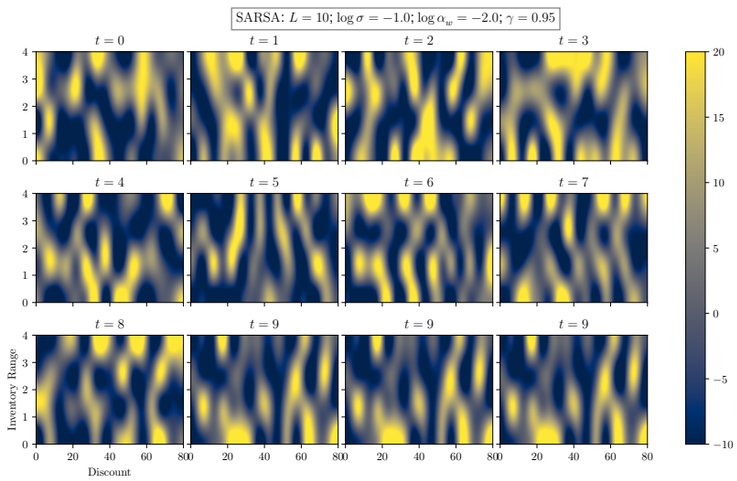




What's interesting about these is that we can see the return achieved at every step (the little orbs around each point), and that they tend to follow fairly uniform reward distributions at each state as

the trajectories make their way into the upper left corner of the "cube" of the state space.

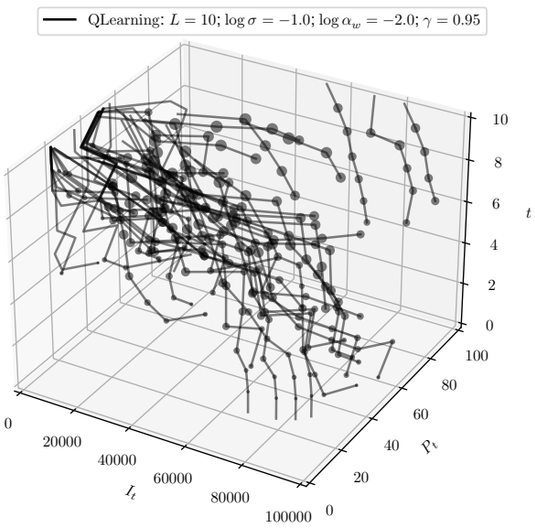
What's interesting here is that a very good majority of the trajectories do in fact make their way towards the lower bounds for both price and inventory, with a lot of trajectories ending up at our lowest price point of 20% of the initial price, and a lot of trajectories making their way to 0. What we can glean from this is that the agent is in fact learning to optimize its actions in any state, however this appears to be offset by a lot of the seemingly random trajectories. In the 2D projection of the 3D trajectory graph, we can see these in the rightmost graph, with the trajectories towards the top. A lot of these are doing seemingly random actions, likely because of the fact that the state space is so large and the transitions are stochastic, it's possible that the agent isn't able to effectively figure out the best things to do in every possible state, especially given that it does not have the option to go back in time. We can somewhat see this when we look at the policy generated as well:

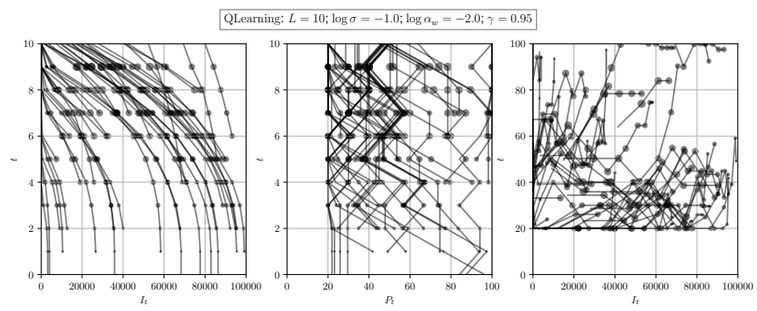


what we see here is that there isn't much cohesion to the policy pattern between timesteps until the later values of t. Specifically, the early times have very different probabilities of actions taken, and as a result, the early states are likely to have high variance in where the trajectory leads, which could probably explain the poor performance of SARSA on this MDP.

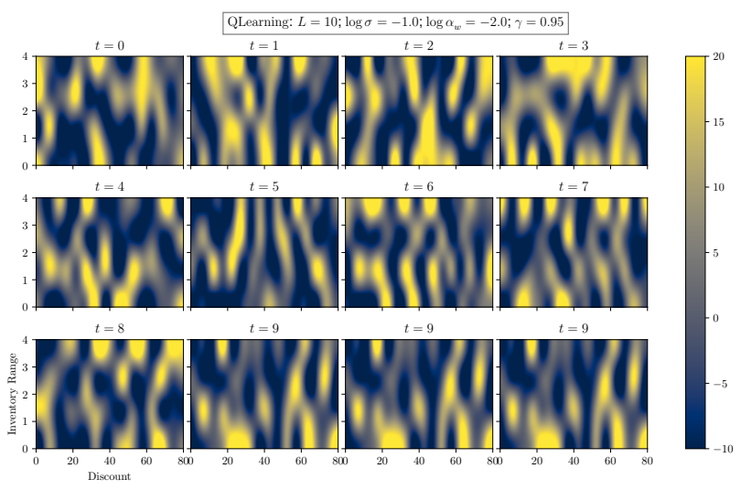
Q-Learning

Given that Q-Learning is very similar to SARSA, the results shown by the agent are not particularly surprising, since they are very similar to the SARSA results. What's of note here, however, is that there is far less variance in the results compared to that of SARSA or the random agent. This can be explained via the pseudocode for Q-learning, where instead of looking at the policy's action in s', it takes $\max_{a \in A} R(s', a)$. As such, whenever the agent is looking at whatever state it ends up in, it'll always choose the action that maximizes the reword. When this is applied alongside random initialization of states, we end up having more consistent results overall, with the occasional peaks and valleys that the learning curve represents. We can glean more information by looking at the trajectories for this algorithm, similar to how we did for SARSA:





What's striking about these trajectories when compared to SARSA is that it looks like a lot of the trajectories are more uniform this time around. We see less random trajectories ending weirdly, and instead the agent is more focused on getting to the top left corner of the cube (which corresponds to the state $(0, 20, t)$). This matches up with the results on the comparison results curve, where on median our Q-Learning algorithm has less variance. Looking at the policy as well:

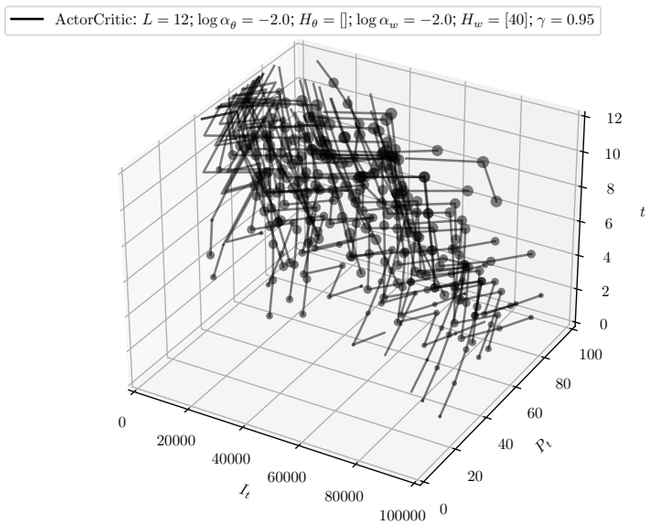


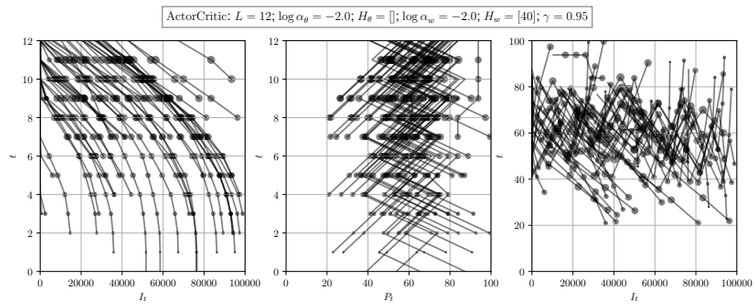
We can see that there are a lot of similar structures across the state space, especially once we hit t=4, and it only gets more solidified as the time goes on, showing once again how Q learning is consistent (even if the actual rewards obtained aren't that much better on average than a random agent)

Actor-Critic

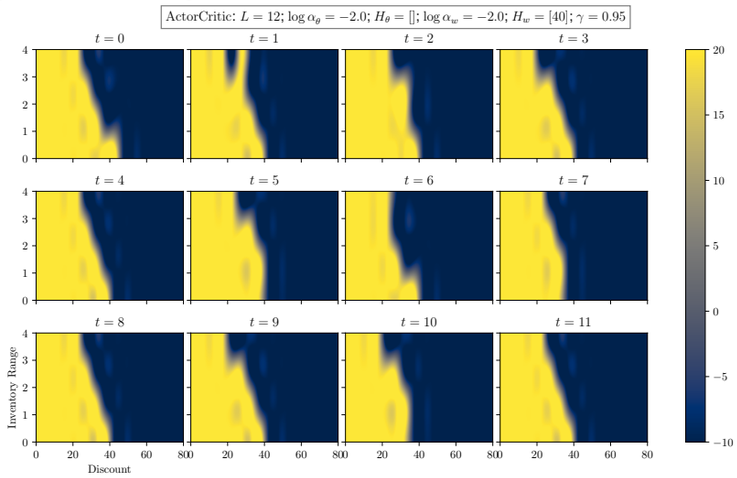
In the case of One-Step Actor-Critic, most of the trajectories that the agent describes converge to $I_L = 0$, which means that the agent is being able to sell all the inventory in the time that it is able to do so. In contrast to tabular agents, the agent tries to set a price that is in the middle of the possible prices. This behavior is related to the trade-off that exists between reducing the price to have more demand and increasing the price to have more retribution per product while having less demand. As

we can deduce from the learning curves in the beginning of the section, there is an optimal point between the minimum and maximum discount.





The sampled trajectories, however, are completely understandable once we look at the policy the agent has learned. For almost any time, the agent tends to increase the discount if the discount is too low and decrease the discount if the discount is too high. This is reflected as the almost-vertical boundary in the next image:



The boundaries are not completely vertical, though. We can observe that for cases where the agent is running out of stock (Inventory Range: 3-4), it lowers the discount more than if it had full stock (Inventory Range: 0-1), since it realizes that the time is being over and the demand will increase because of this. Therefore, it allows itself to set a higher price in the end of the episode than in the beginning.

Conclusion and Potential Improvements

In conclusion, we can see that the problem space of clearance sales as per the fast fashion model is a really complex. While the algorithms we implemented (shy of Actor-Critic) were not the most effective at solving the problem, there is a lot to be learned from this. What we worked with in this project

was a vastly simplified version of the state space and environment dynamics, and did not even consider complicating factors such as size availability, or competition from rivals for the same product. Even on this smaller problem set, however, we notice that the model is still complex enough that methods like SARSA and Q-Learning only barely outperform random algorithms entirely. As such, I think what could potentially be improved is trying different implementations of the state space, like perhaps making the model work disjoint of inventory, instead tying it to price, time, and perhaps one of the other non-considered factors. In addition, we observed that ActorCritic produces the best results amongst the agents we chose, perhaps in future iterations, more gradient-based methods, like TD0 or reinforce with baseline.

Task Division

Vinay Ramesh:

- **MDP**: Formulated the Clearance Sale problem, researched literature and simplified problem space to the MDP, created heuristics for evaluating agent performance.
- **Writing**: Introduction, Broader look at Clearance sales, Quantifying Demand, State Representation, MDP Properties, Results (Metrics, SARSA and Q-Learning Analysis), Conclusion
- **Coding**: `MDP.py` and general debugging.
- **Hyperparameter Tuning**: SARSA and Q-Learning

Ignacio Gavier:

- **MDP**: Provided feedback to the general idea.
- **Algorithms**: Selection of suitable algorithms for the MDP.
- **Writing**: Algorithms, Hyperparameter tuning, Results (Actor Critic analysis).
- **Coding**: Directory structure. Files `train_model.py`, `sarsa_qlearning.py`, `tabularq.py`, `actorcritic.py`, `actor.py`, `critic.py`, `hyperpar_lab.py`, `generate_learningcurve.py` and general debugging.
- **Hyperparameter Tuning**: Actor-Critic